26/01/2024
Friday

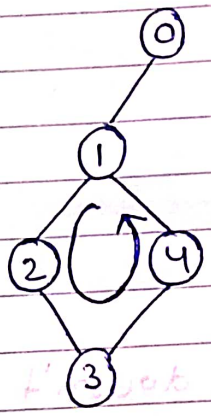
* **Detect cycle in an undirected graph -**

Given an undirected graph with V vertices labelled from 0 to V-1 and E edges, check whether it contains cycle or not. Graph is in the form of adjacency list where adj[i] contains all the nodes ith node is having edge with.

I/P - V = 5 , E = 5
adj = {{1}, {0,2,4}, {1,3}, {2,4}, {1,3}}

O/P - 1

Explanation -



adjacency list
0 → {1}
1 → {0, 2, 4}
2 → {1, 3}
3 → {2, 4}
4 → {1, 3}

0 → 1 → 2 → 3 → 4 → 1

During traversal, vertex 1 is repeated, it means, there is a cycle.

logic - If we are traversing a particular node, not from its parent node but other than it, it means there is a cycle.

If any node is visited before, we don't have to revisit it. for that, we will keep track of visited nodes.
We will also keep the track of the parent nodes. Parent node is a node through which we are traversing a particular node for the first time.

Here, using BFS, we will detect the cycle.

- ```
  if (nbr == parent){
      continue;
  }
  ```

for instance → 0 —— 1

0 → {1}
1 → {0}

} This doesn't represent a backedge in an undirected graph.

- ```
  if (!visited[nbr]) {
      //push it in queue
      // visited mark
      // set parent
  }
  ```

- ```
  if (visited[nbr]) {
      //cycle present
  }
  ```
  ```

// cycle detection in an undirected graph using BFS

Code -

```
class Solution {
public:

bool detectCycle(int srcNode, unordered_map<int, bool>
                 & visited, vector<int> adj[]){

    queue<int> q;
    unordered_map<int, int> parent;

    q.push(srcNode);
    visited[srcNode] = true;
    parent[srcNode] = -1;

    while(!q.empty()){
        int frontNode = q.front();
        q.pop();

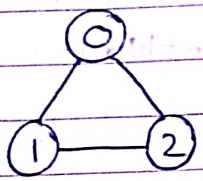
        for(auto nbr: adj[frontNode]){
            if(nbr == parent[frontNode]){
                continue;
            }
            if(!visited[nbr]){
                q.push(nbr);
                visited[nbr]=true;
                parent[nbr] = frontNode;
            }
            else if(visited[nbr]){
                return true;
            }
        }
    }
    return false;
}
```

```
bool isCycle(int V, vector<int> adj[]){

    unordered_map<int, bool> visited;

    for(int node=0; node<V; node++){
        if(!visited[node]){
            bool ans = detectCycle(node, visited, adj);
            if(ans) return true;
        }
    }
    return false;
}
};
```

for instance,



adjacency list	visited	Parent
0 → {1,2}	0 → ~~F~~ T	0 → -1
1 → {0,2}	1 → ~~F~~ T	1 → 0
2 → {0,1}	2 → ~~F~~ T	2 → 0

queue

~~0~~	~~1~~	2	

frontNode = 1

parent[frontNode] = 0

nbr = 2

As 2 is already visited by node 0. Now, we are trying to revisit it from node 1 which is not its parent. from this, came to know, there is a cycle.

Using the same logic, if any node is trying to revisit a particular node which is not its parent, it means there is cycle. We can detect cycle using DFS traversal also.

Code -

```
class Solution{
public:

bool detectCycleUsingDFS(int srcNode, unordered_map
  <int,bool>& visited, int parent, vector<int> adj[])
{
    visited[srcNode]=true;

    for(auto nbr: adj[srcNode]){
      if(nbr==parent){
          continue;
      }
      if(!visited[nbr]){
       bool ans= detectCycleUsingDFS(nbr,visited,srcNode,
                                     adj);

       if(ans) return true;
      }
      if(visited[nbr]){
          return true;
      }
    }
  return false;
}
bool isCycle(int V, vector<int> adj[]){
   unordered_map<int,bool> visited;
   for(int node=0; node<V; node++){
      int parent=-1;
      if(!visited[node]){
       bool ans=detectCycleUsingDFS(node,visited,parent,adj
       if(ans) return true;
   }}
  return false; }};
```

* **Detect cycle in a directed graph-**

Given a directed graph with V vertices and E edges and an adjacency list, check whether it contains cycle or not.

I/P - V = 3 E = 3
adj = {{1}, {0,2}, {}}

O/P - 1

Explanation -

adjacency list

0 → {1}
1 → {0, 2}

0 → 1 → 0 is a cycle.

Logic - If there is a back edge present in graph, it means there is a cycle. Back edge is an edge from a node to one of its ancestors.

To keep the track of already visited nodes, we will use a map here.
To keep the track of current path, we are traversing in graph, we will use dfsTrack i.e. a separate map for that. While going back

from a path, we will backtrack by marking the node false in dfsTrack map.

adjacency list	visited	dfsTrack
0 → {1}	0 → ~~F~~ T	0 → ~~F~~ T
1 → {0, 2}	1 → ~~F~~ T	1 → ~~F~~ T
	2 → F	2 → F

```
if (visited[nbr] && dfsTrack[nbr]){
        //cycle present
}
```

dfs(0)

dfs(1)

dfs(0) → visited[0] is true. Also, dfsTrack[0] is true. Therefore, cycle is present.

Code-

```
class Solution{
public:

bool detectCycleUsingDfs (int srcNode, unordered_map
    <int, bool >&visited, unordered_map<int, bool>&dfsTrack,
    vector <int > adj[]) {

    visited [srcNode]= true;
    dfsTrack [srcNode]= true;

    for (auto nbr : adj.[srcNode]) {
        if (!visited [nbr]) {
            bool ans= detectCycleUsingDFS (nbr, visited, dfsTrack,
                                           adj);
            if(ans) return true;
        }
        else if (visited [nbr] && dfsTrack [nbr]) {
                return true;
        }
    }
    //backtrack
    dfsTrack [srcNode]= false;
    return false;
}
bool isCyclic (int v, vector <int> adj[]){
    unordered_map <int, bool> visited;
    unordered_map <int, bool> dfsTrack;
    for (int node=0; node< v; node++){
        if (!visited [node]){
            bool isCyclic = detectCycleUsingDFS (node, visited,
                                                 dfsTrack, adj);
        if (isCyclic) return true;
}} return false; } };
```